CPE
Redefinição de tipos, Constantes, Registros

Departamento de Engenharia Elétrica – UnB

Roteiro

1. Redefinição de tipos
2. Definição de constantes
3. Registros

Redefinindo um tipo

- Às vezes, por questão de organização, gostaríamos de criar um tipo próprio nosso, que faz exatamente a mesma coisa que um outro tipo já existente.
- Isso é útil quando desenvolvemos programas grandes onde a alteração do tipo de uma determinada variável para outra acarretaria na alteração de muitas variáveis.
- Por exemplo, em um programa onde manipulamos médias de alunos, todas as variáveis que trabalhassem com nota tivessem o tipo `nota`, e não `int` ou `float`.

O comando typedef

- A forma de se fazer isso é utilizando o comando `typedef`, seguindo a estrutura abaixo:

```
typedef <tipo_ja_existente> <tipo_novo>;
```

- Ex: `typedef float nota;`
  Cria um novo tipo, chamado nota, cujas variáveis desse tipo serão pontos flutuantes de precisão simples.

Exemplo de uso do typedef

```
#include <iostream>
using namespace std;

typedef float nota;

int main () {
  nota p1;

  cout << "Digite a nota 1\n";
  cin >> p1;
  cout << "A nota 1 foi " << p1 << "\n";

  return 0;
}
```

Constantes

Frequentemente, utilizamos um determinado valor diversas vezes durante um programa. Para ilustrar, vamos imaginar um programa que trabalhe com um vetor ou uma matriz.

- Na declaração de um vetor limitamos o seu tamanho a um determinado valor máximo;
- Na leitura e na escrita é comum lermos todos os elementos do vetor;
- Em geral percorremos o vetor uma ou mais vezes durante o nosso programa.

Em todos esses casos, utilizamos um mesmo valor para limitar o laço `for` que percorrerá todo o vetor (ou matriz) e para a declaração.

Constantes

- Uma forma de utilizarmos uma única representação para esse valor é declarar uma variável e atribuir um valor constante a ela, mas isso só resolve o problema dos laços, não o da declaração do vetor.

```
int max=20;
int pesos[max]; ⟵ Não faça assim!!!
```

Constantes

- Para resolver isso, utilizamos a diretiva `#define`, que permite definir constantes dentro do programa que podem ser utilizadas em qualquer lugar onde uma constante seria utilizada (inclusive na declaração de vetores e matrizes).

  Ex: `#define MAX_ELEMENTOS 10`

  define a constante `MAX_ELEMENTOS` com o valor 10

Constantes

- O formato padrão da diretiva `#define` é

```
#define NOME_DA_CONSTANTE <valor_da_constante>
```

- Qualquer tipo de constante pode ser colocada no lugar <valor_da_constante>, como pontos flutuantes, inteiros, caracteres, cadeias de caracteres (strings), etc...

Constantes

- A diretiva `#define` deve ser utilizada SEMPRE abaixo da(s) diretiva(s) `#include`, nunca dentro da função `main()`.
- Tipicamente utilizamos letras maiúsculas para o nome das constantes e letras minúsculas para o resto do programa (nome de variáveis, comandos, etc...).

Constantes

- Uma constante não tem tipo, ela não equivale a uma variável. Na verdade, um pré-processador verifica todos os lugares onde você usou a constante e substitui pelo valor à direita ANTES de realizar a compilação. Esse processo é chamado de pré-compilação.
- Uma boa forma de organizar o seu programa é colocar constantes no começo dele, com nomes claros. Logo ao abrir o seu código será possível identificar os limites de seu programa.

Exemplo de uso do #define

```
#include <iostream>

#define NOTA 10
#define MENSAGEM "Parabens, nota "

int main () {
  std::cout << "Parabens, nota " << 10 << std::endl;
  sdt::cout << "Parabens, nota " << NOTA << std::endl;
  std::cout << MENSAGEM << NOTA << std::endl;
  return 0;
}
```

Veja os programas define.cpp e triangular-define.cpp

Registros

Um registro é uma estrutura de dados que agrupa diversas variáveis (chamadas de membros), usualmente de tipos diferentes, mas que dentro de um determinado contexto fazem sentido se agrupadas. Podemos comparar um registro com uma ficha que possui todos os dados sobre uma determinada entidade, por exemplo:

- Registro de alunos (nome, matrícula, médias de provas, médias de labs, etc...)
- Registro de pacientes (nome, endereço, histórico de doenças, etc...)
- Registro de contatos (nome, telefone, aniversário, etc...)
- Catálogo de filmes (nome, ano, duração, atores, etc...)

Declarando o formato do registro

A primeira parte da criação de um registro é declarar seu formato. O formato é como se fosse a declaração do "esqueleto" de um novo tipo de variável. Isso é feito utilizando a palavra chave `struct`, da seguinte forma:

```
struct nome_do_tipo {
  tipo_1 nome_membro_1;
  tipo_2 nome_membro_2;
  tipo_3 nome_membro_3;
  ...
  tipo_n nome_n;
};
```

Declarando o formato do registro

- A declaração de um novo tipo de struct é feita fora da função, como indicado abaixo:

```
#include <iostream>

   /* Declare o formato de seu registro aqui */

int main () {
   /* Construa o resto do programa aqui */
}
```

Declarando uma variável registro

A próxima etapa é declarar uma variável do tipo da struct, que será usada dentro de seu programa, como no exemplo abaixo:

```
#include <iostream>

struct ficha {
  int matricula;
  float media; };

int main () {
  ficha fichapessoa;
  /*resto do codigo aqui */
}
```

Utilizando os membros de um registro

Podemos acessar individualmente os membros (campos) de um determinado registro como se fossem variáveis normais, utilizando a seguinte estrutura

```
nome_do_registro.nome_do_campo
```

Utilizando os membros de um registro

- Para o registro declarado anteriormente, utilizaríamos

  `fichapessoa.matricula`

  para acessar o campo matricula do registro fichapessoa.
- Podemos utilizar o campo de um registro em qualquer lugar onde utilizaríamos uma variável.

Lendo dados para os membros de um registro

A leitura dos membros de um registro a partir do teclado deve ser feita campo a campo, como se fossem variáveis independentes.

```
cout << "Digite a matricula do aluno: ";
cin >> fichapessoa.matricula;

cout << "Digite a media do aluno: ";
cin >> fichapessoa.media;
```

Imprimindo os membros de um registro

A impressão do valor dos membros de um registro deve ser feita campo a campo, como se fossem variáveis independentes.

```
cout  << "O aluno " << fichapessoa.matricula
      << " tirou media " << fichapessoa.media << endl;
```

Exemplo: leitura_escrita.cpp

```cpp
#include <iostream>

using namespace std;

struct ficha {
  int matricula;
  float media;
};

int main (){

  ficha fichapessoa;

  cout << "Digite a matricula do aluno: ";
  cin >> fichapessoa.matricula;

  cout << "Digite a media do aluno: ";
  cin >> fichapessoa.media;

  cout  << "O aluno " << fichapessoa.matricula
        << " tirou media " << fichapessoa.media
        << endl;
  return 0;
}
```

Copiando registros

A cópia de um registro pode ser feita como se fosse a cópia de uma variável normal, ou seja

```
registro_1 = registro_2;
```

Exemplo: copia.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  struct ficha {
6    int matricula;
7    float media;
8  };
9
10 int main (){
11
12   ficha f, g;
13
14   cout << "Digite a matricula do aluno: ";
15   cin >> f.matricula;
16
17   cout << "Digite a media do aluno: ";
18   cin >> f.media;
19
20   g = f;
21
22   cout  << "O aluno " << g.matricula
23         << " tirou media " << g.media << endl;
24   return 0;
25 }
```

Vetor de registros

Pode ser declarado quando necessitarmos de diversas cópias de um registro (por exemplo, para cadastrar todos os alunos de uma mesma turma).

- Para declarar: `ficha fichasturma[5];`
- Para usar: `fichasturma[indice].campo;`

Exemplo: vetor.cpp

```
1  #include <iostream>
2  #define TAM 5
3
4  struct ficha {
5    int matricula;
6    float media;
7  };
8  using namespace std;
9
10 int main (){
11
12   ficha fichasturma[TAM];
13   int i;
14
15   for (i = 0; i < TAM; i++) {
16     cout << "Digite a matricula do " << i+1 << "o aluno: ";
17     cin >> fichasturma[i].matricula;
18
19     cout << "Digite a media do " << i+1 << "o aluno: ";
20     cin >> fichasturma[i].media;
21   }
22
23   for (i = 0; i < TAM; i++) {
24     cout  << "O aluno " << fichasturma[i].matricula
25           << " tirou media " << fichasturma[i].media << endl;
26   }
27
28   return 0;
29 }
```

Registros aninhados

Pode-se também declarar um registro como um dos membros de um outro registro, quantas vezes isso for necessário.

```
struct medias {
  float p1;
  float p2;
  float p3;
};

struct ficha {
  string nome;
  int matricula;
  medias provas;
};
```

Veja o exemplo em `aninhado.cpp`

Apontadores para registros

- Para acessar os elementos de um registro através de um apontador, devemos primeiro acessar o registro e depois acessar o campo desejado ⇒ uso de parênteses.
- Os parênteses são necessários pois o operador * tem prioridade menor que o operador .

Exemplo

```
struct Ponto {
    double x;
    double y;
    };

Ponto *ap_p;

(*ap_p).x = 4.0;
(*ap_p).y = 5.0;
```

Apontadores para registros

- Para simplificar o acesso aos campos de um registro através de apontadores, foi criado o operador `->`.
- Usando este operador acessamos os campos de um registro diretamente através do apontador.

Exemplo

```
struct Ponto { double x; double y; };

Ponto *ap_p;

ap_p->x = 4.0;
ap_p->y = 5.0;
```

Veja o exemplo em `ap_struct.cpp`.

Registros em funções

- Registros podem ser passados como parâmetros de uma função, como qualquer outro tipo.
- O registro deve ser declarado antes da função.
- O parâmetro formal recebe uma cópia do registro, da mesma forma que em uma atribuição envolvendo registros.
- Uma função pode retornar um registro, que é novamente copiado como resultado da expressão.

Exemplo: registro.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  struct ficha {
5    int matricula;
6    float media;
7  };
8
9  ficha le_ficha() {
10   ficha fic;
11   cout << "Digite a matricula do aluno: ";
12   cin >> fic.matricula;
13   cout << "Digite a media do aluno: ";
14   cin >> fic.media;
15   return fic;
16 }
17
18 //const impede que f seja alterado dentro da funcao: fica somente leitura
19 void escreve_ficha(const ficha& f) {
20   cout  << "O aluno " << f.matricula
21         << " tirou media " << f.media << endl;
22 }
23
24 int main () {
25   ficha g;
26   g = le_ficha();
27   escreve_ficha(g);
28   return 0;
29 }
```

Dilbert


I HIRED THE DOGBERT TECHNOLOGY HOSPICE SERVICE TO EASE THE SUFFERING OF OUR DYING TECHNOLOGY.
DogbertCartoonist@gmail.com
Dilbert.com
DOGBERT WILL USE COMPASSION AND... WHAT WAS THE OTHER THING?
BAZOOKA.
5-4-10 ©2010 Scott Adams, Inc./Dist. by UFS, Inc.
STEP AWAY FROM THE WINDOWS XP!